## Directed Sensor Antennae Orientation Algorithm

The algorithm used for orienting the directed sensors was extracted from section 4 (*Approximating the minimum radius*) from the paper *Communication in Wireless Networks with Directional Antennae* by Ioannis Caragiannis, Christos Kaklamanis, Evangelos Kranakis, Danny Krizanc and Andreas Wiese and then converted into code that simply orients the sensors. The coded algorithm uses the same predefined radius and sector angle for all nodes and does not try to determine an optimum radius or an optimum angle.

The algorithm takes in a set of vertices V and computes the minimum spanning forest on it, but it only uses one tree of the forest to make its computations. This would mean that only one connected subgraph is computed, but by iterating through the connected sets of V we are able to effectively orient all subgraphs regardless of whether or not they are in the first tree we saw. At a high level, the operations we perform are as follows: we compute the minimum spanning tree (MST); we use the MST to determine a matching; any vertex not in the matching is oriented towards its parent; every vertex in every couple in the matching orients itself with its coupled neighbor to cover the maximum area they can and then they make connections to any vertices they can reach (in a strongly connected graph, this last step is when the communication to the leaves is enabled). Below we talk a little more in depth about how these are accomplished.

Computing the MST was quite straightforward thanks to the JGraphT library. There is a class *KruskalMinimumSpanningTree* (KMST) to whose constructor you pass the graph on which to make the MST. We pass in a new *ProximityGraph* created using V and are returned an object of the type KMST. Then we populate a new *SimpleGraph* (which we treat as our MST) with all of the vertices in V and all of the edges in the edge set returned by the KMST object.

The pain of creating a matching was also eased, once again, by the JGraphT library. There is a class *BreadthFirstSearchIterator* (BFSI) to whose constructor is passed the graph on which we want to traverse and the start vertex which is treated as the root of the tree. To this class we pass our MST as the graph and as the start vertex, we get a pseudo-random one by calling *next()* on our V iterator. As per the algorithm, the next step we take is adding a random edge connected to the root and we simulate that by calling *next()* on the BFSI and then asking the MST for the edge connecting them. That edge is the first one added to our matching. We keep track of the vertices we’ve used to compute the matching so we add the first edge’s vertices to this *verticesAdded* set. Why do we keep track of the ones we’ve used? That ties in with the next step. The next step is looping through all of the vertices to be visited by the BFSI where, for each iteration, we check if the vertex being examined is a leaf (by examining the number of edges connected to it) or not and act accordingly. If it is not a leaf, if none of its edges (and consequently it) have been added to the matching, we grab the set of edges connected to this vertex and iterate through it until we find an edge where the neighbor vertex has not been added to the matching where we then proceed to add the edge to the matching and record these vertices as having been used. Note that this allows a leaf to be part of the matching if the vertex used an edge connected to a child who is a leaf. If the vertex being examined is a leaf and it has not been used in the matching, we add it to our set of leaves.

One of the products of creating the matching and running the BFS is that we’re left with a set of leaves that have not been used in the matching. The next step in the algorithm is to orient these leaves towards their parent in the MST. This is where we actually start adding edges to our *TransmissionGraph*. This is pretty simple to do, although the math for figuring out the orientation angle can get a bit messy. In any case, if this leaf has an edge (it wouldn’t if the tree isn’t connected), we grab the vertex associated with the other edge, create a directed edge *from* the leaf *to* the parent and add it to the *TransmissionGraph* edge set. Then we have to orient the leaves so that the middle of their transmitted sector is aimed at the parent vertex.

The first step to finding the orientation angle is to find the height and stride (difference between the x-coordinates) difference *from* the vertex being oriented *to* the vertex being ‘pointed to’. The arctangent of this ratio will only give you a value between -90 and 90 degrees but depending on which quadrant we found the angle for (quadrant 1 has both delta height and delta stride positive, quadrant 2 has delta height positive and delta stride negative, quadrant 3 has both delta values negative and quadrant 4 has negative delta height and positive delta stride), the final orientation angle this represents is calculated differently. If we’re in quadrant 1, we can just use the calculated angle as the orientation angle; if we’re in quadrant 2, the orientation angle will be the calculated angle plus 180 degrees (since the angle is negative this gives us a result between 90 and 180 degrees); if we’re in quadrant 3, the orientation angle is the calculated angle plus 180 degrees; if we’re in quadrant 4, the orientation angle will be the calculated angle plus 360 degrees (since the angle is negative this gives us a result between 270 and 360 degrees).

The final step in the algorithm is to finish strongly connecting the MST. This is accomplished by: looping through every edge in the matching, orienting the two vertices of that edge so that they include their coupled neighbor at the extremity of their sector (done by setting the orientation to be the computed angle between the vertices as described above and then adding half of our sector angle to this result) and adding the two edges connecting this coupling to the TransmissionGraph edge set; then, looping through all of our vertices, if the vertex being examined is not one of the ones for the matching edge we’re looking at, if the non-coupled vertex lies in the sector area of either of the coupled vertex, add an edge to our TransmissionGraph going from the respective coupled vertex to the non-coupled vertex (this is where a leaf that wasn’t in our matching would get an edge connected to it). By the end of the iteration of the matching edges, if the graph was able to be strongly connected, it is oriented such that it is.

## Performance Comparisons

The shortest path comparison value represents an averaged ratio over the two graphs for the given trial. The way it’s computed is as follows: compute the set of all of the pairs in the power set of the set of vertices and call it P; for all sets S in P (they all have two sensors) get the hop count of the shortest path between the sensors in S for the Transmission Graph and the Proximity Graph; compute the ratio “Transmission Graph hop count” divided by “Proximity Graph hop count”; add this ratio to the result (which started at zero); once we’ve finished iterating through P, divide the result by the size of P.

The length of routes comparison value also represents an averaged ratio over the two graphs for the given trial. Its result is also initialized to zero and ends up being computed in parallel to the shortest path comparison value (i.e. during the same looping) as follows: for all sets S in P, compute the sum of the Euclidian distance of the shortest path between the sensors in S for the Transmission Graph and the Proximity Graph; compute the ratio “Transmission Graph route length” divided by “Proximity Graph route length”; add this ratio to the result; once we’ve finished iterating through P, divide the result by the size of P.

Computing the network diameter ratio is a lot simpler thanks to the JGraphT library. To compute the network diameter ratio we create two instances of the class *FloydWarshallShortestPath*, one with the Transmission Graph passed into the constructor and one with the Proximity Graph passed into the constructor. We then call *getDiameter()* on both of those instances and divide the Transmission Graph diameter by the Proximity Graph diameter.

It should also be stated that computing the hop count and the shortest paths in these graphs were also done for us by the *FloydWarshallShortestPath* class.

## Logging

For each test we run, we log the data of the sensors in our vertex set, the individual/intermediate results of the shortest path and route length comparisons and finally the average shortest path ratio, average route length ratio and the average network diameter ratio.

## Testing

We didn’t do full-scale, professional testing on our program with formal test cases. Instead, as we implemented features we tested to make sure the new features worked and the old features were not broken. In that regard we did some kind of black box testing. The main testing to do was making sure the implemented orientation algorithm worked since that is what this assignment was really about. That being the case, we’ve come up with some basic cases to test against. Note that, thanks to JGraphT’s *StrongConnectivityInspector*, we are informed in the Graphical User Interface (GUI) whether or not our directed graph is strongly connected.

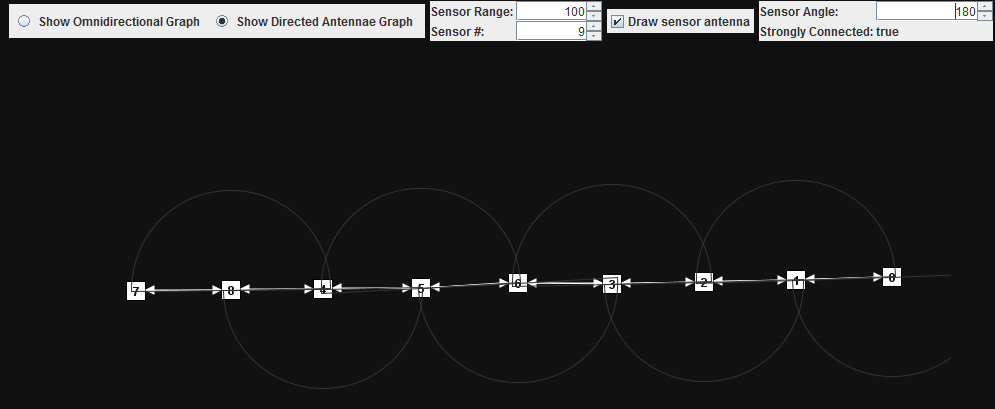


Figure : As we can see, when the sector angle is 180 degrees and they are aligned near perfectly straight, the algorithm orients the antennae such that they form a strongly connected graph

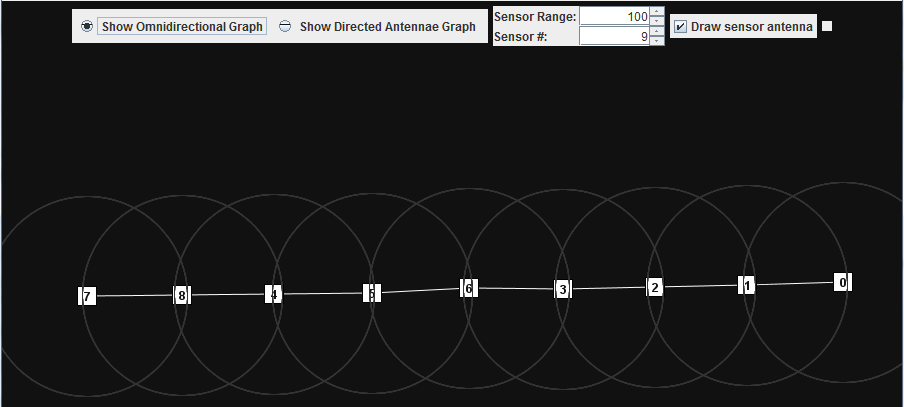


Figure : The omnidirectional version of the graph shown in Figure 1

